1 Introduction

Depth first search is a very useful technique for analyzing graphs. For example, it can be used to

- Determine the connected components of a graph.
- Find cycles in a directed or undirected graph.
- Find the biconnected components of an undirected graph.
- Topologically sort a directed graph.
- Determine if a graph is planar, and find an embedding if it is.
- Find the strong components of a directed graph.

If the graph has n vertices and m edges then depth first search can be used to solve all of these problems in time $O(n+m)$, that is, linear in the size of the graph. In this lecture we will develop some theory about DFS on directed graphs, and show how to solve cycle finding, topological sort, and strong components.

2 Depth First Search in Directed Graphs

Let $G = (V, E)$ be a directed graph, where V is the vertex set and E is the edge set. We assume the graph is represented as an adjacency structure, that is, for every vertex v there is a set $adj(v)$ which is the set of vertices reachable by following one edge out of v. To do a depth first search we keep two pieces of information associated with each vertex v. One is the depth first search numbering $num(v)$, and the other is $mark(v)$, which indicates that v is currently on the recursion stack.

Here is the depth first search procedure:

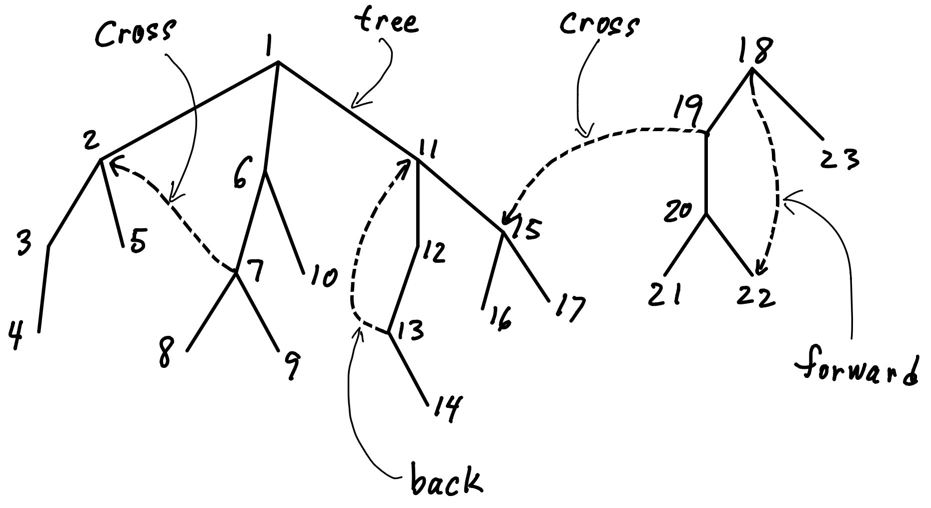
```
$i \leftarrow 0$
for all $v \in V$ do $num(v) \leftarrow 0$
for all $v \in V$ do $mark(v) \leftarrow 0$
for all $v \in V$ do
    if $num(v) = 0$ then DFS($v$)

DFS($v$)
    $i \leftarrow i + 1$
    $num(v) \leftarrow i$
    $mark(v) \leftarrow 1$
    for all $w \in adj(v)$ do
        if $num(w) = 0$ then DFS($w$)          [$(v, w)$ is a *tree* edge]
        else if $num(w) > num(v)$ then         [$(v, w)$ is a *forward* edge]
        else if $mark(w) = 0$ then             [$(v, w)$ is a *cross* edge]
        else                                   [$(v, w)$ is a *back* edge]
    $mark(v) \leftarrow 0$
end DFS
```

This process examines all the edges and vertices. The call DFS(v) is made exactly once for each vertex in the graph. Each edge is placed into exactly one of four classes by the algorithm: tree edges, forward edges, cross edges and back edges.

This classification of the edges is not a property of the graph alone. It also depends on the ordering of the vertices in *adj*(v) and on the ordering of the vertices in the loop that calls the DFS procedure. The *num* and *mark* fields are not actually necessary to accomplish a complete search of the graph. All that is needed to do that is a single bit for each vertex that indicates whether or not that vertex has already been searched. (This bit is zero for vertex v if and only if $num(v) = 0$.) We have presented the fully general version here because it is needed for the strong components algorithm that we present later in this lecture.

The tree edges have the property that either zero or one of them points to a given vertex. (It's the edge used to discover a vertex for the first time.) Therefore, they define a collection of trees, called the *depth first spanning forest* of the graph. The root of each tree is the lowest numbered vertex in it (the one that was searched first). These rooted trees allow us to define the ancestor and descendant relations among vertices.

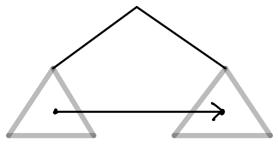


The figure above illustrates a spanning forest with two trees. Whenever we draw pictures of DFS spanning forests we draw the ancestors up and the descendants down. We also draw the tree edges out of a node left to right in the order in which they appear in the adjacency list.

So to summarize:

- **tree edges** form a forest of trees.
- **forward edges** are from a vertex to a descendant in the tree.
- **cross edges** are from node a to node b where the subtrees rooted at a and b are disjoint.
- **back edges** are from a vertex to an ancestor.

Note that it is never possible for there to to be "reverse cross edges", that is, an edge that goes from left to right between two disjoint subtrees, as in this picture:



This is because if such an edge existed, then it would be a tree edge. The vertex on the right end of it would be found while exploring the subtree on the left (which is searched first cause it is drawn to the left), and it would *be in* the subtree on the left.

3 DFS in DAGS (Directed Acyclic Graphs)

Note that a graph has a cycle if and only if it has a back edge. If you look at the picture of the DFS tree with cross edges, forward edges, and tree edges, you see that there is no way to follow edges from any vertex v and get back to v. On the other hand, if there is a back edge, there is clearly a cycle. Follow the back edge from node v to one of its ancestors, then follow tree edges from that ancestor back to v.

We can make a special version of DFS to determine if the graph has a cycle. We don't need the numbers other than to determine if we've already visited a node. (So the *num*() fields can be replaced by a *visited* bit.)

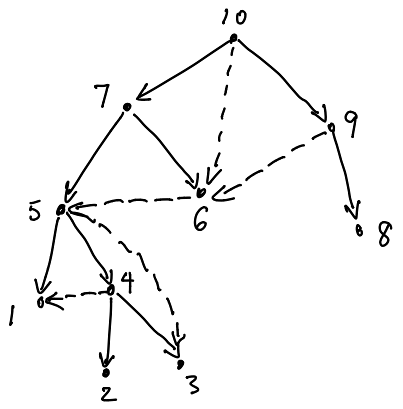
Suppose the graph is acyclic. In this case there exists what is called a *topological ordering* of the vertices. This is a list of all the vertices of the graph such that for any edge (u, v) in the graph the vertex u comes before v in the ordering. A reverse topological ordering is one whose reversal is a toplological ordering. So for any edge (u, v), v occurs before u in the reverse topological ordering.

The following algorithm computes a reverse topological ordering of a graph.

```
for all $v \in V$ do $visited(v) \leftarrow false$
for all $v \in V$ do TopSort($v$)

TopSort($v$)
    if not $visited(v)$ then
        $visited(v) \leftarrow true$
        for all $w \in adj(v)$ do TopSort($w$)
        output $v$
end TopSort
```

Here's an example of what happens when this algorithm is run on a DAG. The vertex numbered 1 is output first, etc.



As usual we draw tree edges as solid, and cross and forward edges as dashed. There are no back edges.

Lemma 1 *When the above algorithm is run on a DAG it outputs a reverse topolological ordering.*

Proof: It suffices to prove that if there exists an edge (v, w) then the algorithm outputs w before it outputs v. And this result follows if the call to TopSort(w) returns before the call to TopSort(v) returns.

Since the graph is acyclic there are no back edges, so (v, w) must be a cross, tree, or forward edge. If it's a cross edge, then the call to TopSort(w) has completed before TopSort(v) even started. If it's a tree edge or forward edge, then the sequence of events is that we initiate the call to TopSort(v), during which we discover w and call TopSort(w), then we complete the call to TopSort(w) then we complete the call to TopSort(v). ■

4 Strong Components

Two vertices v and w of a graph G are *equivalent*, denoted $v \equiv w$, iff there exists a path from v to w in G, and there exists a path from w to v in G. The relation "$\equiv$" so defined is an equivalence relation because it satisfies the following three properties.

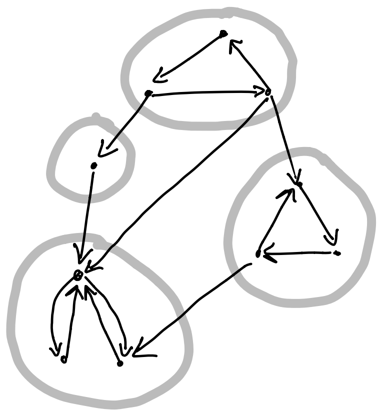
Reflexivity: $w \equiv w$. This is because a path of length zero goes from w to w and vice versa.

Symmetry: If $v \equiv w$ then $w \equiv v$. This follows immediately from the definition.

Transivity: If $u \equiv v$ and $v \equiv w$ then $u \equiv w$. If there is a path from u to v and from v to w then there is one from u to w. The same reasoning shows there is a path from w to u.

This equivalence relation induces a partitioning of the vertices of the graph into components in which each pair of vertices is a component is equivalent. These are called the *strongly connected components* (or sometimes just *strong components*) of the graph. Our goal is to devise an algorithm which will compute the strong components of a graph.

Here is an example of a directed graph partitioned into its strong components.



Note that the strong components (when regarded as vertices) form a DAG. If there were a cycle in this graph, then all the components on this cycle would be part of the same strong component.

5 Determining if a Graph is Strongly Connected

A graph is said to be *strongly connected* if it has one strongly connected component. Now we will give a DFS-based algorithm to determine if a graph is strongly connected. This is a stepping stone to our ultimate goal of finding the strong components.

Say a graph G is strongly connected. What must happen when you run DFS on it?

- There will be one tree in the spanning forest rooted at vertex number 1.
- There must be a way of getting from any vertex back to vertex number 1.

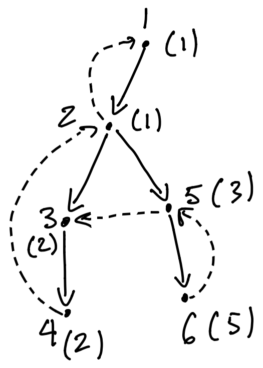
In fact these two properties of the DFS also imply that G is strongly connected.

What can we compute as we go along that would be useful for this? Idea: *lowpoint* numbers:

lowpoint(v) = The lowest numbered vertex reachable from v using zero or more tree edges followed by at most one back or cross edge.

This is defined in a way that is both easy to compute, and such that it can be used to determine strong connectivity. The reason it's easy to compute recursively in the DFS is that it only requires following *one* back or cross edge. It *does not* require that we compute the lowest numbered vertex reachable by any path. We'll see how this makes it easy to compute in a second.

The figure below shows a graph which has been explored by DFS. The tree edges are solid and non-tree edges are dashed. The DFS numbers are shown on each vertex, and the *lowpoint* numbers are shown in parentheses.



Algorithm L for computing *lowpoint* numbers:

Do a DFS on the graph starting from an arbitrary vertex called v_0. When we visit a vertex v for the first time we assign *lowpoint*(v) to be *num*(v). When considering edge (v, w) if it is a tree edge we do DFS(w) then we do the assigment

$$lowpoint(v) \leftarrow \min(lowpoint(v), lowpoint(w))$$

If it's a cross edge or back edge we do the assignment

$$lowpoint(v) \leftarrow \min(lowpoint(v), num(w))$$

Lemma 2 *The above algorithm correctly computes the* lowpoint *numbers for all vertices reachable from v_0.*

Proof: The proof is by induction on the DFS tree. When processing v the first thing we do is call DFS on all of its children. These calls compute the *lowpoint* values of these nodes. We assume inductively that it has worked on the children. Now using this assumption we prove that it holds for v.

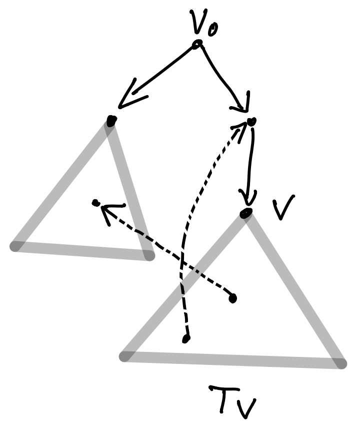
Consider the path from v resulting in its *lowpoint* value. This could start with a tree edge to one of its children. In this case we will compute *lowpoint*(v) correctly by induction. On the other hand if it starts with a back or cross edge, then it applies the appropriate update rule *lowpoint*$(v) \leftarrow \min($*lowpoint*$(v), num(w))$ for this path of length 1. ∎

Lemma 3 *The graph is strongly connected if and only if (a) Everything is reachable from vertex v_0 and (b)* lowpoint$(v) < num(v)$ *for all* $v \in V - v_0$.

Proof:

$\Leftarrow$ By (a), there is a path from v_0 to every other vertex. From statement (b) we know that for all $v \in V - v_0$ there exists a path to a vertex v' with with $num(v') < num(v)$. If $v' \neq v_0$ then there is a path to a v'' with $num(v'') < num(v')$. We can continue to concatenate these paths to lower and lower numbered vertices. Eventually we must end up at the vertex v_0 whose *num* is 1.

$\Rightarrow$ If the graph is strongly connected then (a) holds trivially. Now consider the path from $v \neq v_0$ to v_0. This path must leave T_v (the subtree of the DFS spanning forest rooted at v) for the first time. Consider that edge. It must be a back or cross edge to a lowered numbered vertex. This proves that *lowpoint*$(v) < num(v)$, and property (b). See this figure:



∎

Later we will make use of the following corollary, which summarizes what we have discovered about *lowpoint* numbers.

Corollary 4 *If the graph is strongly connected, then for all vertices v except v_0 (the starting vertex of the DFS) we have* lowpoint$(v) < num(v)$. *For v_0 we have* lowpoint$(v_0) = num(v_0)$

6 Computing the SCCs Using Base Vertices

Consider the vertex numbering produced by running DFS on a graph G. Each strongly connected component (SCC) of G has some lowest numbered vertex in it. Call that the *base* of the SCC.

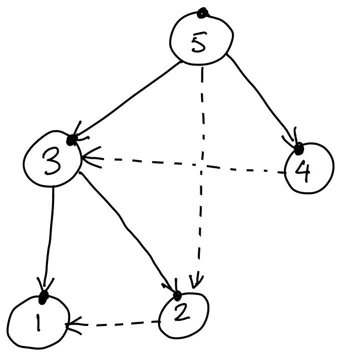
Suppose we magically knew for each vertex if it was a base vertex of some SCC. Then the DFS can be augmented to compute the SCC of the graph. Here's how this is done.

Algorithm B for computing SCCs using base vertices:

Do a complete DFS as explained in section 2. Maintain a stack of vertices. When we first visit a vertex (i.e. assign it a positive number) we put it on the stack. Whenever we complete the DFS(v) function for a vertex v that is a base vertex we bulk pop vertices off the stack until v is popped off, then we stop popping. Each such group of popped vertices is a SCC.

Lemma 5 *The above stack-augmented DFS algorithm using knowledge of the base vertices, computes all the SCCs of the graph.*

Proof: The proof is based on studying the picture below. Each circle represents a SCC of the graph. As usual the solid edges are tree edges and the dashed ones are cross or forward edges. There cannot be any back edges between these components. The number shown on each component is its reverse topological order. As we will see, the SCCs will be popped out in reverse topological order. So 1 is the first one to come out, 2 is next, etc. The highlighted dot in each components is its base vertex.



Let's study what happens over time. First some of the vertices in component 5 are visited, then it moves to component 3 (where some are visited), then it moves to component 1. Eventually it explores all of SCC 1. Here's a snapshot of the stack at this point in time: [All of SCC 1 are on the top portion of the stack. SCC 1's base vertex is the deepest of these. Then come some from component 3, after which there are some from component 5.] It will then complete the call to DFS on the base vertex of SCC 1. At which point it will pop out all of component 1.

It then goes back to 3 for a while and eventually moves on to 2. Although there is an edge from 2 to 1, no vertices from 1 ever get on the stack again, because they were already visited. Eventually 2 is fully explored and exited, at which point it gets popped off the stack.

Although this is just one example, it's completely generic and covers all the cases. The key fact is that at each point in time when it is about to pop the stack, (1) all the vertices from the current component occupy a contiguous region on the top of the stack (the base vertex is the deepest on the stack from that SCC), and (2) all of the vertices of all the SCCs earlier in the reverse topological order have already been popped. So it ends up just popping out one complete component. ■

7 Computing the Base Vertices

The trick to finding the base vertices is to extend our concept of *lowpoint* to the more general framework of non-strongly-connected graphs. For purposes of clarity we give these numbers a different name: *lowlink*.

lowlink(v) = The lowest numbered vertex *in the same strong component as* v reachable from v using zero or more tree edges followed by at most one back or cross edge.

The only difference between this definition and that of *lowpoint* is in the emphasized phrase "*in the same strong component as v*".

Now we need to do two things. We first need to explain how to find the base vertices given that we know the *lowlink* values. Secondly we need to explain how to compute the *lowlink* values. The following lemma gives us a way to identify the base vertices.

Lemma 6 *A vertex is a base if and only if* $num(v) = \mathrm{lowlink}(v)$.

Proof: The idea is that the *lowlink* numbers play the same role in one SCC as the *lowpoint* numbers played in a strongly connected graph. So we will be able to apply Corollary 4 to prove this lemma.

Let C be one of the SCCs of the graph G. C is simply a subgraph of G whose vertices and edges are those within the chosen SCC.

Let $v \in C$ be a vertex. Let $num(v)$ be the number of v computed by the DFS in G. Let $num'(v)$ be the number of v computed by the DFS in C in isolation, starting at the base vertex of C. These two DFSs compute the same edge classifications within C, and visit the vertices within C in the same order. This immediately implies that the relative ordering of a pair of vertices with respect to num and num' are the same.

In other words, given two vertices v and w in C we have that $num(v) < num(w)$ if and only if $num'(v) < num'(w)$. The same holds for the $=$ and $>$ relations.

The definitions of *lowlink* and *lowpoint* are the same except that the paths considered when computing *lowlink* are allowed to go out of the component C. But a path which exits C can never get back into C, because it's a strong component. So the only paths that the *lowlink* definition can make use of are those that stay inside C. In other words the set of allowed paths for both the *lowlink* computation and the *lowpoint* computation are the same.

It immediately follows that for any vertex v in C, $lowpoint(v) < num'(v)$ if and only if $lowlink(v) < num(v)$. And $lowpoint(v) = num'(v)$ if and only if $lowlink(v) = num(v)$. From Corollary 4 we know that the starting vertex v_0 of the DFS in C is the only one for which $lowpoint(v) = num'(v)$. Thus it follows that the base vertex of C is the only one in C that has the property that $lowlink(v) = num(v)$ ■

We'll now modify Algorithm B to eliminate the need for it to have the base vertices. It computes them by computing the *lowlink* numbers as it goes along. This is a complete algorthm for finding the SCCs of a graph.

Tarjan's SCC Algorithm:

Do a complete DFS as explained in section 2. Maintain a stack of vertices. When we first visit a vertex v compute $num(v)$ in the usual way, and assign $lowlink(v)$ to be $num(v)$. We also put v on the stack. Then we call DFS recursively on w for each tree edge (v, w). This will compute $lowlink(w)$. Upon the return of DFS(w) we udpate $lowlink(v)$ as follows:

$$lowlink(v) \leftarrow \min(lowlink(v), lowlink(w)).$$

If (v, w) is a cross edge or back edge, and w *is on the stack* we do the assignment:

$$lowlink(v) \leftarrow \min(lowlink(v), num(w)).$$

When we complete the DFS(v) function we test if $lowlink(v) = num(v)$. If so, v is a base vertex, so we bulk pop vertices off the stack until v is popped off, then we stop popping. Each such group of popped vertices is a SCC.

Theorem 7 *Tarjan's SCC Algorithm computes the strongly connected components of the graph.*

Proof:

We've already shown that if we have the *lowlink* numbers we can find which vertices are the base vertices. (Lemma 6). We've also shown that if we have the base vertices the algorithm will find the SCCs of the graph (Lemma 5). So it only remains to prove that this algorithm correctly computes the *lowlink* numbers.

The proof will make use of induction based on the ordering of vertices in the DFS spanning forest. So we'll assume that the *lowlink* numbers of all the children of v have been correctly computed after completing their calls to DFS.

Let's define the notion of a *lowlink path* from v This is a path that begins at v and follows zero or more tree edges, then follows zero or one cross or back edge and ends up at a vertex x in the same strong component as v. We have defined $\mathit{lowlink}(v)$ to be the minimum vertex number reachable from v via a lowlink path.

Let's consider the first edge (v, w) of a lowlink path for v. This could be a tree edge, a back edge, or a cross edge. Let's consider all three cases.

tree edge: Let's suppose that w is in the same SCC as v. In this case continuing the lowlink path from w will give us a valid lowlink path for v. And the algorithm considers this.

Suppose that w is not in the same SCC as v. In this case there is no valid continuation for the lowlink path of v. This is because once a path leaves v's SCC it can never return to it. It is also the case that w is a base vertex in its SCC. So $\mathit{lowlink}(w) = \mathit{num}(w) > \mathit{num}(v) \geq \mathit{lowlink}(v)$, and the assignment $\mathit{lowlink}(v) \leftarrow \min(\mathit{lowlink}(v), \mathit{lowlink}(w))$ cannot have any effect on $\mathit{lowlink}(v)$. So this path is effectively ignored, as it should be.

(Note that although the algorithm does not do the stack check in the case of a tree edge, it could do so without changing the value of any $\mathit{lowlink}()$ number. The implementation at the end of these notes makes use of this simplification.)

cross edge: Let's suppose that w is in the same SCC as v. As in the previous case w must be on the stack, because v is on the stack. Therefore the algorithm correctly considers $\mathit{num}(w)$ as an option for $\mathit{lowlink}(v)$.

If w is not in the same SCC as v, then the edge (v, w) is a cross edge to a different SCC. As explained above there cannot be a valid lowlink path in this case. As in the proof of Lemma 5, we know that w's SCC was popped off before we even began to search v. Therefore w is not on the stack, and the algorithm correctly does not consider these paths.

back edge: In this case w is an ancestor of v. Therefore there's a cycle containing both w and v, so they are in the same SCC. Since v is on the stack w must also be on the stack by the bulk popping argument mentioned above. Therefore the algorithm correctly considers this lowlink path (consiting of one back edge).

This completes the proof that *lowlink* is computed correctly, and that the algorithm computes the SCCs of the graph. ■

8 Implementation

Below is an "implementation of this algorithm" in Java. I put that in quotes because this is actually a little bit simpler than the algorithm described in the previous section.

The difference is that it does not quite implement the update rule for *lowlink* numbers described above. Instead it computes what are called *low* numbers, by applying the update rule designed above for tree edges to all kinds of edges. This means that the code does not have to distinguish between different kinds of edges which simpifies it a little (i.e. no *mark*() bits are needed).

So in the end, this code does not compute the *lowlink* numbers described above. However the numbers it computes, *low* numbers, are possibly smaller (but no greater) than the *lowlink* numbers. But this reduction cannot cause a *low* number to decrease below the lowest *num* in the same strong component. This means that the test for a vertex being a base vertex $low(v) = num(v)$ remains valid.

```java
import java.util.*;
import static java.lang.Math.min;

public class TarjanSCC {
    int n, count, comp;
    int[] num, low, answer;
    boolean[] onStack;
    Stack<Integer> stack;
    int[][] graph;

    public int[] strong (int[][] g) {
        graph = g;
        n = graph.length;
        num = new int[n];
        low = new int[n];
        answer = new int[n];
        onStack = new boolean[n];
        stack = new Stack<Integer>();
        count = 0;
        comp = 0;
        for (int x=0; x<n; x++) DFS(x);
        return answer;
    }

    void DFS(int v) {
        if (num[v] != 0) return;
        num[v] = low[v] = ++count;
        stack.push(v);
        onStack[v] = true;
        for (int w: graph[v]) DFS(w);
        for (int w: graph[v]) if (onStack[w]) low[v] = min(low[v], low[w]);
        if (num[v] == low[v]) {
            while (true) {
                int x = stack.pop();
                onStack[x] = false;
                answer[x] = comp;
                if (x == v) break;
            }
            comp++;
        }
    }
}
```